プログラミング言語 Polemy の設計と実装

http://www.kmonos.net/repos/polemy

メタプログラミングの会 2010

しゃべる人　k.inaba

あるいは

メタプログラマは

2度走る

メタプログラマは n度走る

Polemy って何ですか？

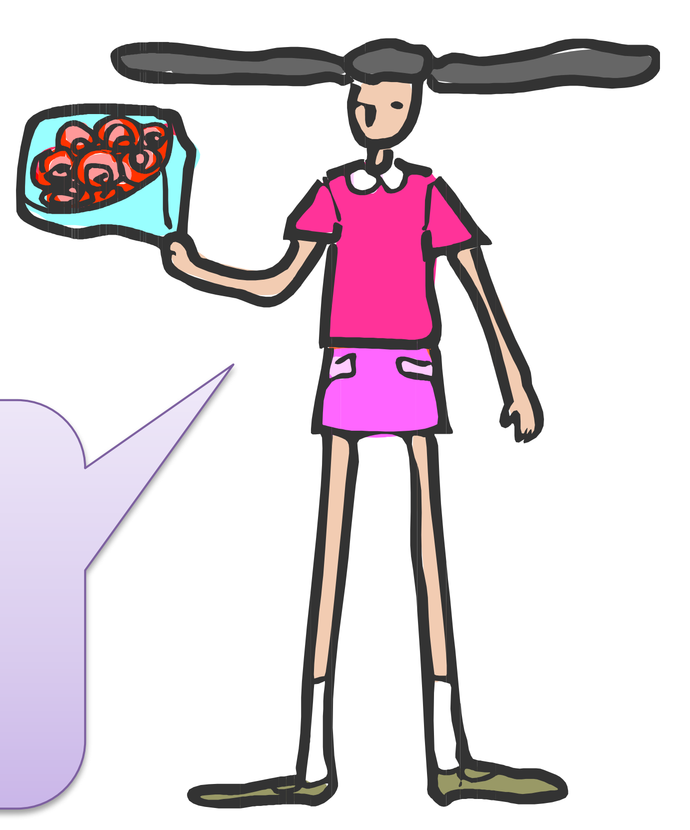
Polemy って何ですか

先月できた新しい言語です

[作者] k.inaba

[目的] 今日話すネタを作るため

```
$ ./polemy
Welcome to Polemy 0.1.0
>>
```



ポレミちゃん(仮)

何系の言語？

関数型？論理型？
オブジェクト指向？

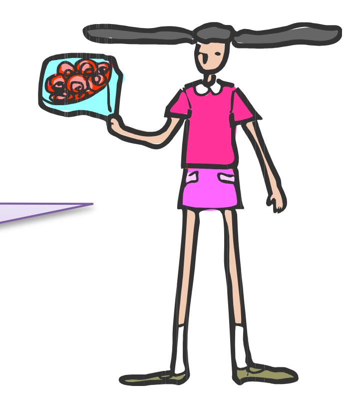
Twitter / @poremi: @...
twitter.com/poremi
B!
twitter
検索
ホーム
プロフィール
メッセージ
@poremi
Polemy Girl
@macro times(n @value, e) { @value(if n == 1 then @macro(e) else @macro(e; times(n-1,e)))}; times(99, print("Hello, World!"));
12月6日 Twit for Windowsから
お気に入りから外す
返信
削除

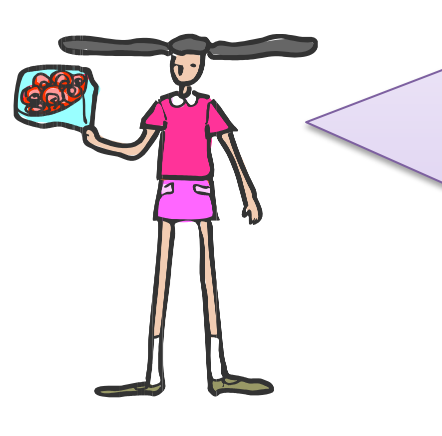
Twitterに貼ると reply飛びまくる 系言語

冗談はさておき

大部分は普通の言語です

```
print("Hello, World");
```





```
def pow(x, n) {
  if n==0 then 1 else
   ( var y = pow(x*x, n/2);
     if x%2==0 then y else y*x )
}
```

FAQ

- 静的型システムは？

“ない” - “です”

- オブジェクトは？ JS や Lua 的に…

```
var obj = {table: “が”, aru: “ので”}; obj.ganbare
```

- 第一級の関数は？

```
let K = fun(x){fun(y){x}} in  K(“あります”)(“よ”)
```

- パターンマッチは？

```
case lst when {car: x, cdr: xs}: “ある”
```

- 破壊的代入は (作者怠慢のため) まだない

本題

Meta Programming

in Polemy

発想の元は

- 型レベルプログラミング
  - in <u>**C++**</u>, Haskell, Scala, D, …

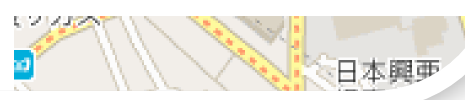
イベント開催支援ツール：ATND（アテンド）

Not logged in

型レベルプログラミングの会

Types as Programming Languages

日時 / DATE : 2009/04/18 12:00 to 18:00



例: C++

```
vector<int> xs = …;
  nth(rev_iter(xs.end()), 5);
```

末尾を指すイテレータを計算

+ と – の意味を逆転したイテレータに変換

5 歩進めた先の値取得

どこに
型レベル
計算が？

例: C++

```
vector<int> xs = …;
  nth(rev_iter(xs.end()), 5);
```

vector<int> から int* を計算

int* を reverse_iterator<int*> に変換

int*からintを取得

※注： 正確には、int* とは限りません。vector<int>::iterator です

静的型付け言語では
コードは
２つの意味で
２度走る

２つの視点

```
template<class T>
class vector {
   T arr[]; int siz;
   T* end() {return arr+siz;}
}
template<class It>
iterator_traits<It>::value_type
  nth(It it int n)
    { return *(it+n); }
```

値レベルの視点

```
template<class T>
class vector {
   T arr[]; int siz;
   T* end() {return arr+siz;}
}
template<class It>
iterator_traits<It>::value_type
  nth (It it, int n)
    { return *(it+n); }
```

型レベルの視点

```
template<class T>
class vector {
   T arr[]; int siz;
   T* end() {return arr+siz;}
}
template<class It>
iterator_traits<It>::value_type
  nth(It it, int n)
    { return *(it+n); }
```

- vector<T>::end() は
  - 値レベルでは、arr+siz を計算して返す関数
  - 型レベルでは、T* を返す定数関数

- nth(It it, int n) は、
  - 値レベルでは *(it+n) を計算して返す関数
  - 型レベルでは iterator_traits<It> というサブルーチンを呼び出して、イテレータの指す型を計算する関数

さらに

２つのレベルの交叉

- この場合、nth(it,n) の値レベル動作は
  return *(it+n);

```
vector<int> xs = …;
nth(rev_iter(xs.end()), 5);
```

- この場合、nth(it, n) は
  while(n-->0) ++it; return *it;

```
list<int> xs = …;
nth(rev_iter(xs.end()), 5);
```

型レベル計算の結果で、値レベルの挙動を決める

```
template<typename It>
  iterator_traits<It>::value_type
nth(It it, int n) {
  iterator_traits<It>::ry_category が
    random_access_iterator_tag なら
      return *(it+n);
    forward_iterator_tag なら
      while(n-->0) ++it; return *it;
}
```

ここまでのポイント

- 静的型付け言語では
  1つのコードが2つの意味/計算を表す
  - "Static Semantics" と "Dynamic Semantics"

- C++er や Haskeller はよくご存じですが

これは便利

１つのコードに

semantics

２つの意味

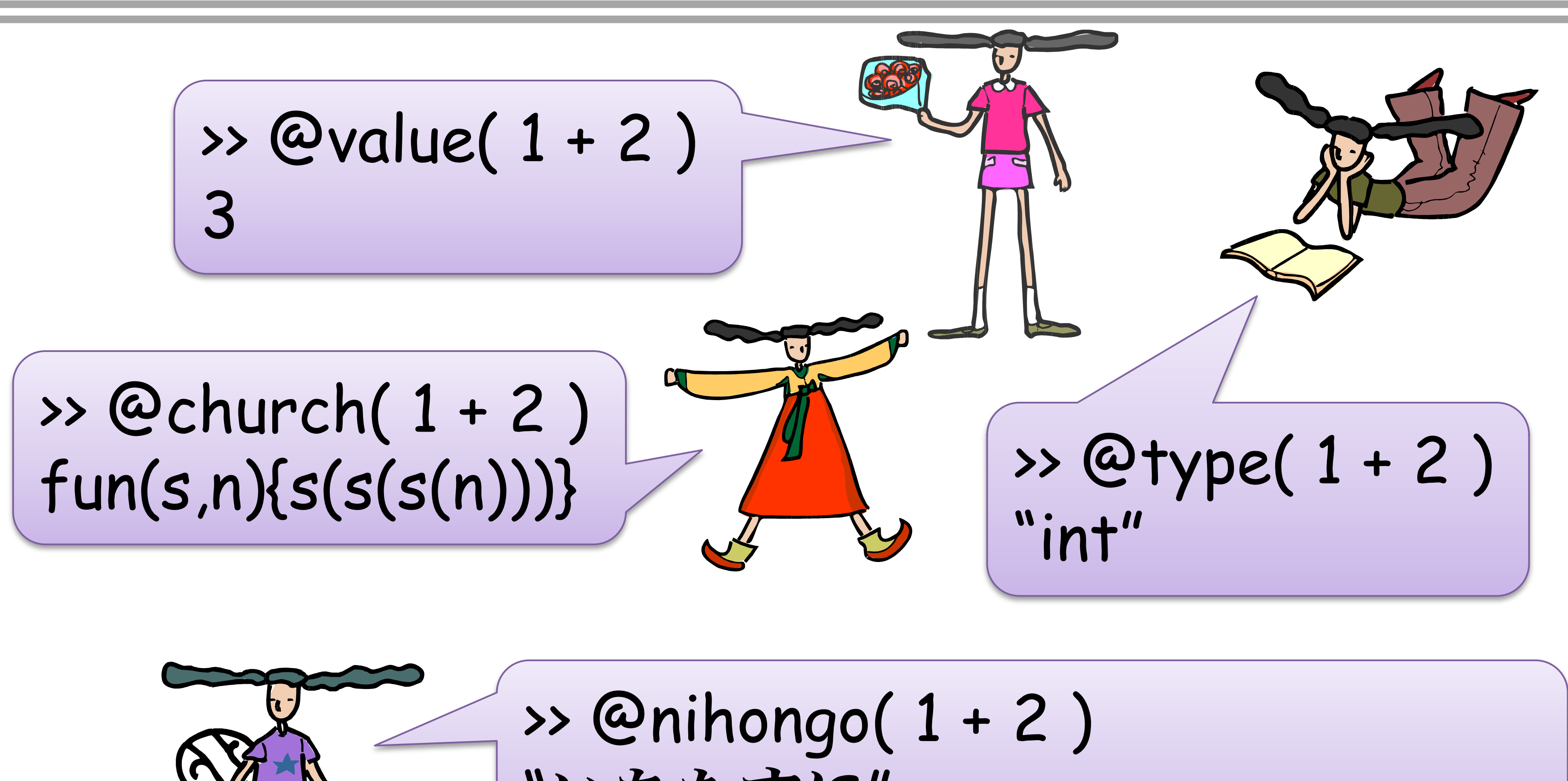
なぜ
２つなのか

３つじゃ
ダメ
なんですか

それと

これ
別に
型の上でやらんでも
いいよね！

そこで Polemy



Polemy って何ですか

- 型レベル計算の本質（と私が思うもの）

  「１つのコードを複数の意味で解釈」

  これだけを取り出した言語

- 本質と思わないもの
  - 型によって
    - プログラムの正しさを保証
    - プログラムを高速化
  - これは型レベル計算でできることの一例
- ぶっちゃけ、型はどうでもいい

レイヤ指定実行

Expr ::= … | “@” LAYERID “(“ Expr “)”

```
>> @value( 1 + 2 )
3
```

- コード 1+2 を「@value レイヤ」で実行
  - “1” や “+” や “2” の意味を「値レベルで」で解釈&実行
  - @value はデフォルトで用意されてる言語組込レイヤです

レイヤ指定実行

Expr ::= … | “@” LAYERID “(“ Expr “)”

```
>> @type( 1 + 2 )
int
```

- コード 1+2 を「@type レイヤ」で実行
  - @type はユーザ定義のレイヤです。頑張って作ります
  - “1” や “+” や “2” の意味を「型レベルで」で解釈＆実行

レイヤの作り方 (1)

- リテラルの「意味」を与える**リフト関数**

```
Expr ::= …
    | “@@” LAYERID “=“ Expr “;” …
```

```
@@type = fun(x){
  if( _isint(x) ) “int“
  else “ERRORRRR!!!!“
};
```

int 専用
超絶手抜き
型検査

レイヤの作り方 (2)

- @type レイヤでは 1 の意味は “int”
- @type レイヤでは 2 の意味は “int”
- @type レイヤでは “a” の意味は “ERRORRRR!!!!”

```
@@type = fun(x){
  if( _isint(x) ) “int”
  else “ERRORRRR!!!!”
};
```

レイヤの作り方(3)

- **レイヤ指定変数定義**で既存変数の意味を

```
Expr ::= …
    | "@" LAYERID VAR "=" Expr ";" …
```

```
var + = <primitive_plus>;
@type + = fun(x,y) {
    if (x=="int" && y=="int")
        "int"
    else "ERRORRRR!!!!"
};
```

レイヤの作り方(3)

- レイヤ指定変数定義で既存変数の意味を

```
Expr ::= …
    | "@" LAYERID VAR "=" Expr ";" …
```

```
var + = <primitive_plus>;
@type + = fun(x,y) {
    if (x=="int" && y=="int")
        "int"
    else "ERRORRRR!!!!"
};
```

レイヤの作り方(4)

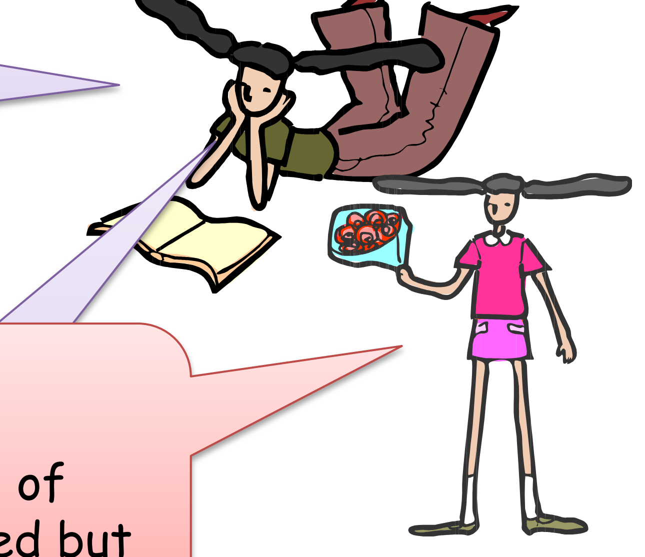
```
@type + = fun(x,y) {@value(
   if ( @type(x)==“int”
   && @type(y)==“int” )
      “int”
   else “ERRORRRR!!!!”
)};
```

できました

```
>> @type( 1 + 2 )
int
```

```
>> 1 + "orz"
[<REPL>:10:1] type mismatch on the argument 2 of native function +. polemy.value.IntValue required but orz passed.
```

```
>> @type( 1 + "orz" )
ERRORRRR!!!!
```



値レベルで
動かさないでも
静的？に型検査

できました

- ユーザ定義関数は勝手に意味を持ちます
  （再帰の処理にはちょっとトリックが必要）
- @valueレベル関数 ➔ @typeレベル関数

```
>> def f(x,y,z) { x + y + y }
>> @type(f(1,2,"orz"))
int
>> @type(f(1,"orz",2))
ERRORRRR!!!!
```

「複数の意味」はできました。

```
>> @type( 1 + 2 )
int
```

```
>> @value( 1 + 2 )
3
```

「１つのコードに複数の意味」は？

```
nth(rev_iter(xs.end()), 5);
```

- **レイヤ指定引数を使います**

```
def next(it @type @value, n) {
  if( @type(it).tag == “RandomAccess” )
    it + n
  else
    if n==0 then it else next(it.next, n-1)
}
next( 1+2, 5 );
```

他の例

たらい
を
回す

無駄に関数を呼びまくる関数

```
def tarai(x,y,z) {
    if x<=y then
        y
    else
        tarai(tarai(x-1,y,z),
              tarai(y-1,z,x),
              tarai(z-1,x,y) )
};
```

```
>> tarai(12,6,0)
```

終わらない

遅延評価がすごく効く関数

まったく同じtaraiのコードを
遅延評価な意味で実行したい

```
def tarai(x,y,z) {
 if x<=y then
   y
 else
   tarai(tarai(x-1,y,z),
         tarai(y-1,z,x),
         tarai(z-1,x,y) )
};
```

```
>> @lazy(tarai(12,6,0))()
12
```

遅延評価がすごく効く関数

```
def t
 if x<
  y
 else
   tarai(tarai(x-1,y,z),
             tarai(y-1,z,x),
             tarai(z-1,x,y) )
};
```

```
@@lazy = fun(x){fun(){x}};
@lazy - = fun(x,y){fun(){@value(@lazy(x)()-@lazy(y)())}};
@lazy <= = fun(x,y ){fun(){@value(@lazy(x)()<=@lazy(y)())}};
@lazy if = fun(c,t,e){fun(){@value(if @lazy(c)() then
                                   @lazy(t())() else @lazy(e())())}};
```

```
>> @lazy(tarai(12,6,0))()
12
```

余談

@joke(Polemy.言語名の由来)

プログラミング言語 P
俺々言語 ole
マイ言語 my

@wikipedia(言語名の由来)

Polemic

From Wikipedia, the free encyclopedia

(Redirected from Polemy)

For other uses, see Polemic (disambiguation).

This article **may require cleanup to meet Wikipedia's quality standards**. Please improve this article if you can. The talk page may contain suggestions. *(March 2010)*

A **polemic** (pronounced /pəˈlɛmɪk/) is a variety of argument or controversy made against one opinion, doctrine, or person. Other variations of argument are debate and discussion. The word is derived from the Greek *polemikos* (πολεμικός), meaning "warlike, hostile".[1]

メタプログラミングの会に
「論争状態」
を巻き起こしたいですね！

@truth(言語名の由来)

multiple meanings wikipedia

About 285,000 results (0.18 seconds)

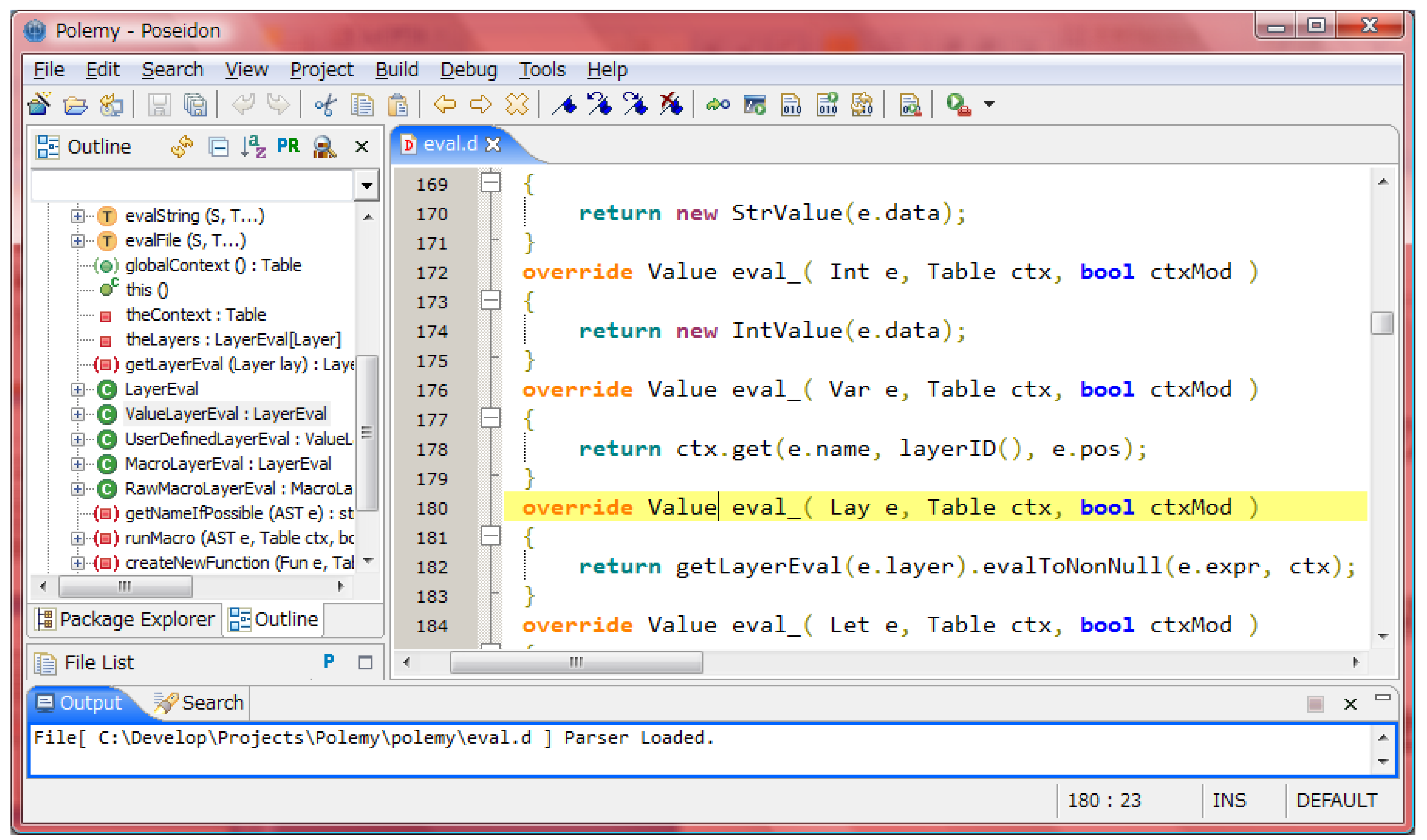
Polysemy - **Wikipedia**, the free encyclopedia

or signs to have **multiple meanings** (sememes), i.e., a larg
word or phrase with multiple, related meanings. ...
en.**wikipedia**.org/wiki/Polysemy - Cached - Similar

長いので、これを
適当に縮めました

実装

D言語でできています



実装

- D のメタプログラミングの発表が無いみたいなので

ちょっとだけ Polemyの実装からメタっぽい部分の紹介
（たいしたことはしてません）

文字列で頑張る

- unittest ライブラリにて
  - 文字列 mixin ≒「コンパイルタイム eval」

```
void assertOp(string op,A,B)(A a, B b) {
  if( !mixin("a"~op~"b") )
    throw new AssertError(text(a," !",op," ", b));
}
assertOp!("==")(123, 456);  // Error: 123 !== 456
```

コンパイル可能性チェック

```
class Token {
  immutable LexPosition pos;
  immutable string      str;
  immutable bool        quoted;
}
assert(
  !__traits(compiles, (new Token).str=“foo") );
```

```
// std.range (本当はis(typeof(…))を使っている)
template isInputRange(R){
  enum isInputRange = __traits(compiles,{
    R r;           auto _ = r.front;
    r.popFront;   if(r.empty){}        });
}
```

型情報

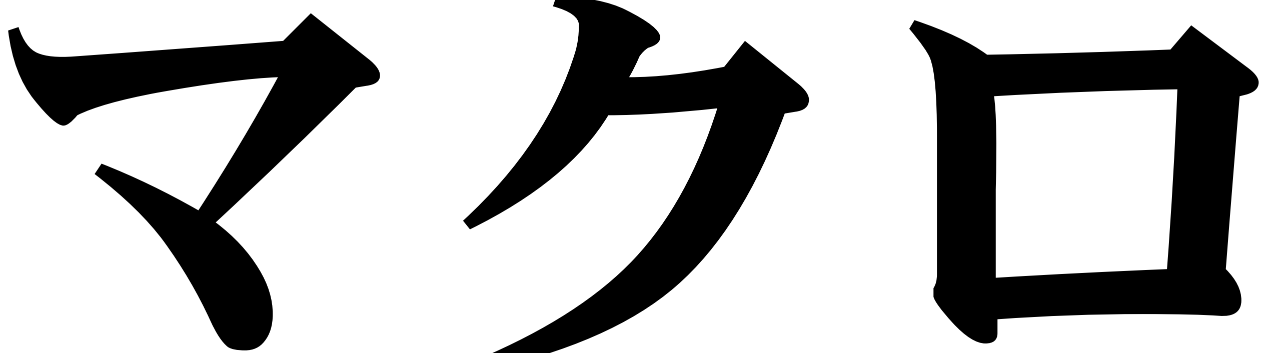
```
e.addPrimitive( “+”, (IntValue a, IntValue b){
  return new IntValue(a.data + b.data);
});
// ↑こう書いたら↓こういう関数を登録
(Value[] args){
   if( args.length != 2 ) throw …;
   if( args[0] is not IntValue ) throw …;
   if( args[1] is not IntValue ) throw …;
   return 上記の関数(
     cast(IntValue)args[0], cast(IntValue)args[1]
   );
}
```

コンパイル時リフレクション

```
T polemy2d(T)(Value v) {
  …  static if(is(T == class))
       foreach(mem; T.tupleof)
         mem = v.get(mem.stringof); …
}

class Hoge { int foo; string bar; }

auto x = polemy2d!Hoge(
  eval(`{foo: 100, bar: "hello"}`)
);
```

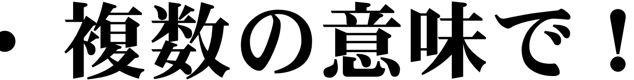

マクロ

プログラムを
データ（構文木）
と見なして
読んだり書いたり

「プログラム」を「構文木」と見る

- 一つのコードを

```
>> 1 + 2
3
```



- 複数の意味で！

```
>> @macro( 1 + 2 )
{ pos:..., is: App,
  fun: {pos:..., is:Var, name:+}
  args: [ {pos:..., is:Int, data:1},
          {pos:..., is:Int, data:2} ] }
```

Polemy の実行ステップ

入力されたソース

@macro レイヤで実行

@value レイヤで実行

とりあえず
「コードを実行するとその構文木を作る」
ように、インタプリタ組み込みで実装

@macroレイヤで実行

その瞬間に
マクロ機能
完成

レイヤ指定変数定義

```
@macro
 LetItBe = fun(b, expr){
    var it = b;  expr
 };

LetItBe(1+2,  print(it));
```

レイヤ指定引数

```
@macro
 rep = fun(n@value, e@macro) {
   @value(
     if n==1 then
       @macro(e)
     else
       (@macro(e); rep(n-1, e))
   )};
rep(100, print("MACRO!!"));
```

それ◯◯で
できるよ

「レイヤ」に
似てるもの大集合

コンテキスト指向プログラミング (COP)

Classbox

__Classboxes__: A Minimal Module Model Supporting Local Rebinding [Bergel&Ducasse&Wuyts 2003]

selector namespace

Refinement

__Ruby 2.0__: Classboxes, nested methods, and real private methods
http://www.slideshare.net/ShugoMaeda/rc2010-refinements

多重世界機構

超時空プログラミングシステム
<u>Uranus</u> [中島 1985]

これらも「根本的に世界を変える」違い。
- lift function でリテラルの意味まで変える
- 基本的には「世界」を継承しない

<u>Worlds/JS</u>
Worlds: Controlling the Scope of Side Effects [Warth 2009]

ActiveS式

***Ziggurat*: Static-Analysis for Syntax Objects [Fischer&Shivers 2006]**

マクロが、単なるS式に展開されるのでなく
- 「展開」メソッドを呼び出すとS式を返す
  open classのオブジェクトになってる
- ノードにメソッド足せる
- 静的解析のためにオーバーライドすると、
  元の式レベルに意味を展開しないで解析が！！

"完全に別の static semantics" を好き勝手
足すという意味ではこれが近い。
自分で定義したマクロ以外の意味も
オーバーライドしたいよね？
というのがPolemy

準クオート

これを @macro と @value 以外に
一般化した感じ

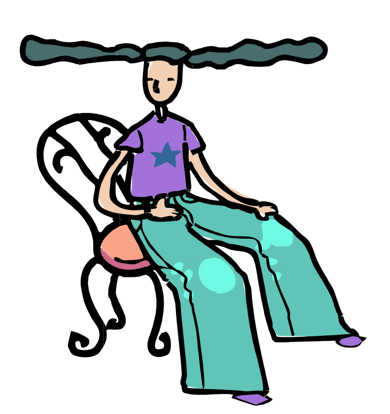
quasiquote ` == @macro
unquote , == @value

scala.reflect.code
C#の式木

もっと色々激しく
「コンテキスト」増やしましょう

Perlの
リスト/スカラ

Pluggable Types



Pluggable Type Systems
[Bracha 2004]

JSR 308: Annotations on Java Types

色んな（静的）セマンティクスを
好きなだけ足そう、というスピリットはここから来ている

C#やJavaの
Annotation

ただ、単に「構文木」を扱うメタプログラム
じゃなくて、「関数 is 関数」「変数 is 変数」
パラダイムで、できるだけ元のプログラムを
そのまま”実行”する形でやりたい

HOAS

Higher Order

Abstract Syntax

構文木を
「関数は関数」で表現

抽象解釈をユーザコードレベルで色々
定義できるようにしたもの、として
Polemy を使えるかも、

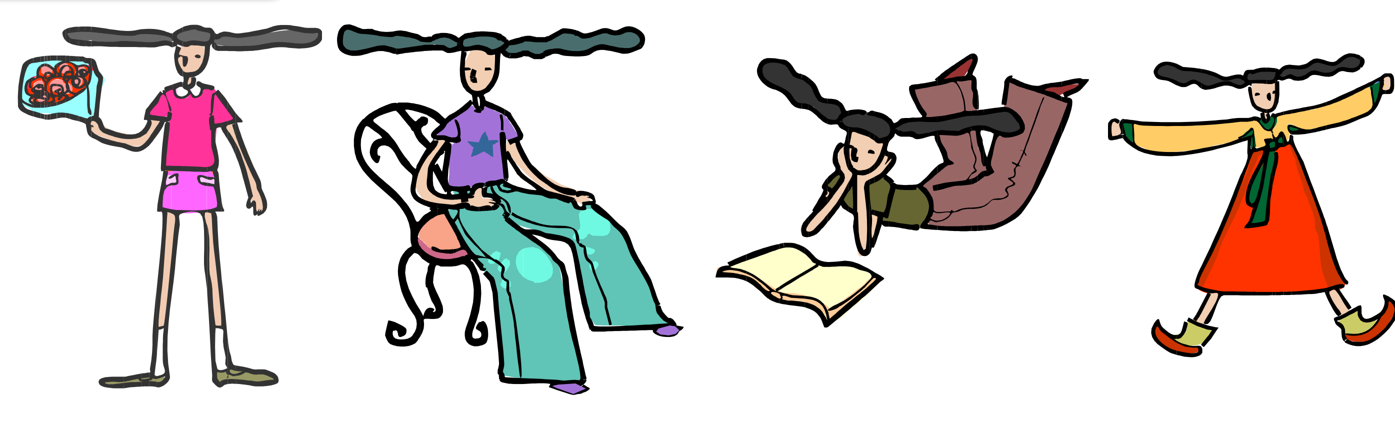
と思ったけど、有限性を利用した停止性を
メモ化だけで実現しようとしているので
かなり無理がある

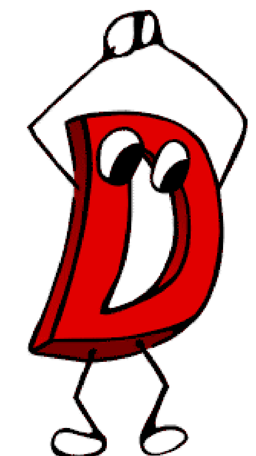
値を抽象化した状態で計算することで、
コードの色んな性質を静的に解析する技法

モナド モナド

おわり

Cast





PowerPoint のクリップアートにいた素敵な髪型の女の子　　名称未設定

URL

http://www.kmonos.net/repos/polemy

Slogan

「1つのコードに複数の意味」

Todo

- 「レイヤを使ったコードを別レイヤで実行する」（≒ @type を使ったコードの @type での型検査）が全然うまくいってない
- 作った値を過去に遡ってレイヤ評価できた方がいいかも（var x = e; @lay(x) で e を見たい）→ 遅延評価？